SYNCTRACE: Visual Thread-Interplay Analysis

Benjamin Karran Jonas Trümper Jürgen Döllner
Hasso Plattner Institute, University of Potsdam, Germany

***Abstract*—In software comprehension, program traces are important to gain insight into certain aspects of concurrent runtime behavior, e.g., thread-interplay. Here, key tasks are finding usages of blocking operations, such as synchronization and I/O operations, assessing temporal order of such operations, and analyzing their effects. This is a hard task for large and complex program traces due to their size and number of threads involved. In this paper, we present SYNCTRACE, a new visualization technique based on (bended) activity diagrams and edge bundles that allows for parallel analysis of multiple threads and their inter-thread correspondences. We demonstrate how the technique, implemented as a tool, can be applied on real-world trace datasets to support understanding concurrent behavior.**

***Index Terms*—Trace analysis, Software visualization, Program comprehension, Concurrency.**

I. INTRODUCTION

In the software engineering lifecycle, software maintenance is a crucial part [13], [28]. Studies report that over 40% of the total maintenance efforts are spent on program comprehension [6]. One key reason for this is the complexity of programs and their behavior, which is particularly true for concurrent programs. Here, in addition to static analysis, dynamic analysis is a valuable tool for program comprehension [5], [15], [45]. Understanding the interplay of threads by (non-)blocking operations in specific scenarios is thereby a key strategy in program comprehension of concurrent programs [7]. While static analysis of a program's code base can only reveal a sound dataset that is valid for all executions of a program [9], dynamic analysis, or *program tracing*, can capture runtime information *(program traces)* relating to specific scenarios during execution [45]. Such recorded traces help explaining used concurrency patterns and thread responsibilities, locate performance bottlenecks as well as root causes of bugs caused by non-determinism and incorrect synchronization. Moreover, such traces can be used to verify/falsify hypotheses about the (mis-)usage of blocking operations in a given program.

Behavior analysis of concurrent programs is difficult for several reasons. Recorded program traces are massive data that pose several analysis and representation challenges [45]. Moreover, we need to show much information per thread, such as call relationships, timestamps, order of synchronization points as well as I/O operations, and related blocking system calls (including wait duration). Due to limited screen space and certain aspects of human cognition, it is hard to present this information for many threads in parallel.

This work was funded by the Federal Ministry of Education and Research (BMBF), Germany within the InnoProfile Transfer research group "4DnD-Vis".

In this paper, we present a visualization technique for the interactive analysis of large traces from concurrent programs. Our visualization design enables the parallel analysis of many threads' runtime behavior and inter-relationships by a hybrid focus+context visualization approach: A combination of straight (focus) and bended (context) activity diagrams depicts intra-thread call relationships, and edge bundles show inter-thread relationships caused by blocking operations, such as synchronization. We further support the exploratory nature of the underlying task and address *visual* scalability by a multiscale design that allows for trace analysis on several levels of abstraction, and complementary specialized interaction techniques.

II. RELATED WORK

Concurrency analysis and understanding is an established research domain since the first parallel computers were introduced, and can be classified as follows.

Activity Views: (Concurrent) runtime behavior, if given as hierarchical sequences, is commonly depicted by variants of icicle plots [21] where the horizontal axis is mapped to time and the vertical axis to 'location', such as stack depth [38] or memory block range [24]. Rotated variants are also used to visualize acyclic object graphs, such as the Java heap [29]. In a similar way, variants of UML sequence diagrams can be used [8], which, however, are less efficient than icicle plots in terms of space usage. Scatter plots, as a multivariate visualization, help correlate items in high-dimensional datasets, such as peer-to-peer download metrics [41] or program traces [30].

Concurrency Analysis: Visualization techniques focus either on multi-machine (distributed) or single-machine (shared-memory) systems. While the two share many concepts, there are important differences – such as the means used for communication and synchronization. Visualization of shared-memory systems typically does not include network communication, but visualization for multi-machine systems does [16], [26], [27], [46]. Moreover, for multi-machine systems, runtime behavior is often analyzed on a coarser scale, i.e., the runtime behavior of single processes or threads is rarely of interest. In the following, we discuss only visualization of shared-memory systems.

Stasko and Kraemer [34] visualize concurrent executions (as call graphs) and synchronization, but do not show call stacks over time as we do. Several visualization support analysis

978-1-4799-1457-9/13/$31.00 ©2013 IEEE

of threaded computing from the scheduling perspective by focusing on thread states (running, suspended etc.) [3], [4], [19], [47]. Several tools use variants or extension of UML sequence or activity diagrams to depict activity and inter-thread communication [1], [23]. While UML diagrams typically allow for detailed analysis [12], [44], they do not scale well for large traces and/or high number of threads: They quickly become unusable due to visual clutter. For example, showing inter-thread relationships between non-adjacent threads often creates many edge crossings.

Correspondence Visualization: Various techniques for depicting correspondences (e.g., correlations or communication) on hierarchical sequences, such as program traces, exist [14]. Several techniques use node-link diagrams [20], [43] to show correspondences in concurrent executions. Such diagrams are generally more prone to visual clutter than other techniques since they cause massive edge crossings. TreeJuxtaposer [25] draws two trees side-by-side and highlights related tree nodes by color and interaction. Holten et al. [18] build upon this idea by connecting two juxtaposed icicle plots (having uniform visual depth) with hierarchically bundled edges [17], which represent the correspondences. The similar CodeFlows [36] technique works on multiple trees with non-uniform depth by placing the trees (drawn as rotated icicle plots) next to each other. By this, the technique can only show correspondences between each two adjacent trees. Beck et al. [2] use a similar approach than CodeFlows, but use it for analyzing multi-dimensional correlations on *one* tree. Artho et al. [1] use extended UML sequence diagrams to show concurrency-specific correspondences between threads. The technique is able to show correspondences between more than two neighbored threads by using overdrawing, which often results in cluttered diagrams for larger traces. Several recent approaches provide advanced line rendering, e.g., by applying ambient occlusion [10] or screen-space line aggregation [48]. Image-based edge bundling [37] and TraceDiff [39] (for hierarchical edge bundles) improve upon the above techniques by implicitly encoding the width of correlated elements using 2D tubes instead of 1D curves to show the correspondences.

Our concurrency-analysis use-case joins these challenges: Visualize both activity (call stacks) and inter-relationships (blocking operations) of multiple threads in parallel, while enabling interactive (and synchronized) manipulation of the shown subsequences and threads. Moreover, due to the cardinality of the trace data, the visualization has to support multiscale analysis of the trace data, and allow for context-preserving transitions between those scales.

III. Trace-Data Processing

We next describe the trace-data model and how we capture this data at runtime. Based on this model, we will subsequently describe our visualization design and related interaction techniques.

A. Data Model

We model a program trace as a set $P = \{T_m\}$ of *thread traces*, where m is considered the *thread id*. A thread trace is in turn a tree $T = \{f\}$ of function calls

$$f = (N, t^s \in \mathbb{R}^+, t^e \in \mathbb{R}^+, p \in T, C, O \subseteq I) \tag{1}$$

with

$$C = \{c_i \in T\},\ I = \{I_j\}. \tag{2}$$

In f, N depicts an identifier of the function call, such as the function name. The call's start and end moments are represented by t^s, respectively t^e, where $t^s < t^e$. Additionally, p is considered the caller of f. If f is the root of T, then p is undefined. The set C contains all calls that f called to, ordered by call time. Further, we distinguish four kinds of calls:

1) S_w is the set of all calls to a synchronization function that waits for the objects in O.
2) S_r is the set of all calls to a synchronization function that releases the objects in O.
3) S_o is the set of all calls to a function that issues I/O requests, i.e. writes or reads a file.
4) S_c is the set of all calls to any other function.

For any f, with $S \notin S_c$, O is a set of objects on which the function call f operates. Hence, the set of all objects in trace P is called I. The objects may be mutexes, semaphores, files, or other objects related to synchronization or I/O.

In addition, we introduce the following functions:

$$\mathrm{dur}(f) = t^e - t^s \tag{3}$$

$$\mathrm{dur}^x(f) = \begin{cases} \mathrm{dur}(f) & \text{if } f \in S_x \\ \sum_{c \in C(f)} \mathrm{dur}^x(c) & \text{otherwise} \end{cases} \tag{4}$$

where x in $\{o, w, r\}$; $\mathrm{dur}(f)$ can be interpreted as the duration of a function call f. Similarly, dur^w, dur^r and dur^o tell the duration that f spent doing synchronization, respectively I/O. Further, the stack depth of f is defined as:

$$\mathrm{depth}(f) = \begin{cases} 0 & \text{if } p(f) = undef \\ \mathrm{depth}(p(f)) + 1 & \text{otherwise} \end{cases} \tag{5}$$

B. Data Extraction

In order to record a program's behavior we use a dynamic binary instrumentation approach. We built a *pintool* using the Pin instrumentation framework [22] that employs function boundary tracing to log function entry and exit events. To be notified upon function exit, Pin instruments all return instructions. However, functions can return from within another function due to compiler optimization. We further instrument all return instructions in a module to avoid missing any exit notifications [11].

To discern synchronization and I/O function calls from other calls, we instrument the operating system's synchronization and I/O API routines. In addition, we evaluate some of the functions' parameters to determine the set O of objects on which the function calls operate on.

IV. Visualization Design

A. Exploration Workflow

With our design, we follow the common understanding of exploration strategies applied in program comprehension: Users applying top-down and bottom-up exploration (with frequent switches between both) and cross-referencing [35], [42]. By a multiscale representation of the trace data, we can, at a coarse scale, identify synchronization patterns and outliers, as well as the concurrency patterns used in a program, such as farmer/worker, leader/followers etc. [32]. At medium and fine scale, by contrast, detailed analysis of thread synchronization is possible.

We start by selecting a number of threads to visualize, each containing hundreds of thousands of events. Next, we see an overview visualization of the trace data, which allows for identifying which are the main synchronization relationships between which threads, which thread T_i waits mostly on which other thread T_j, and who waits longest. Subsequently, we can drill-down (top-down exploration) using zoom&pan to analyze subsets of the trace data in more detail. We can now see where (stack level, function f) and when a thread waits or releases by drawn overlays. Aggregation techniques allow for compressing less interesting activity (stack levels) using simple pan interaction. On the waiting side, we can further investigate the amount of time spent waiting (and further details on demand), and which thread did release the waited object. On the release side, we can assess what happens while the locked object is being waited on. Visual attributes allow for discerning different wait types, such as I/O cycles and synchronization.

At any point in our analysis, we can switch to bottom-up exploration using the provided zoom facilities and directly jump to other subsets of the trace data using overview navigation.

B. Visualization Concept

The visualization is split into three parts: Overview window, main view and context view (Fig. 1). The overview window is placed at the top, and linked with the main view in a focus+context fashion. The main view displays one thread trace at a time as an icicle plot where each bar in the plot has a start and end coordinate (B_s, B_e), as well as a discrete level (Fig. 2). Every function call f of a thread trace T is then mapped to a bar B, where B_s and B_e correspond to t^s, respectively t^e, and depth(f) is mapped to the bar's level. Also, the end B_s may be augmented with a shaded texture, which helps finding repetitive patterns with respect to the duration of function calls.

The context view is placed at the bottom and displays a variable number of thread traces. All thread traces together form a semicircle where each thread trace is assigned an equal section by default. To see which thread trace is visualized in a section, the section is labeled with the thread id as colored circle, each thread T having one color col(T) out of a predefined set of colors. Although such assignment of colors is not bijective, we constrain the assignment such that

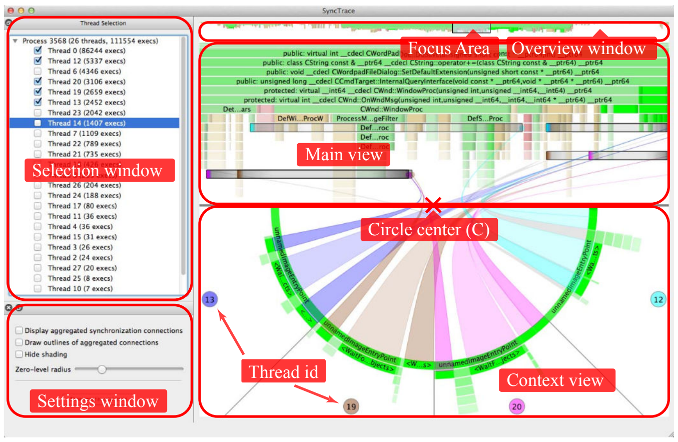


Fig. 1. Overview of the SyncTrace main window.

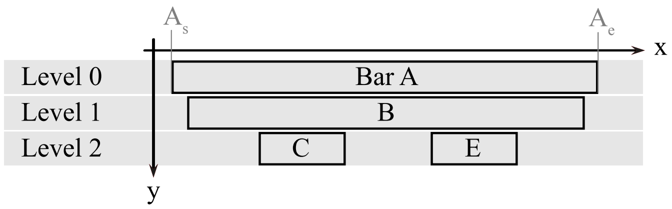


Fig. 2. Bar A has the start coordinate A_s, end coordinate A_e, and level 0.

no two adjacent threads can have the same color. By this, we ensure that repetitions of thread colors can only occur for two spatially distant (in terms of angle on the semicircle) threads.

Like the thread trace in the main view, the thread traces in the context view are displayed as icicle plots. However, in contrast to the main view, the icicle plots are bended (similar to the SunBurst visualization [33]). Therefore, all thread traces in the context view share a common circle center C and radius Z for level 0 (Fig. 1). Instead of mapping the levels to the y-coordinate, we here map them to the radius of a circle centered in C. Level 0 is assigned the zero-level radius Z, and subsequent levels are assigned increasing radii such that the icicle plots grow from the center of the circle to the outside. The start and end coordinates of a bar are mapped to an angle. As a result, deeper stack levels are assigned more space for the same time span. So, although deeper levels often consist of shorter function calls, the additional space on the circumference in effect provides more space for labels. Hence, even short low-level function calls can have readable labels.

1) Correspondences Among Thread Traces: In the area surrounding the circle center, the curves, each connecting two thread traces, represent synchronization correspondences between the threads. Such a correspondence always consists of a function call f_w that waited for a resource and another call f_r that released (or signaled) the resource such that f_w can stop waiting. More precisely, a correspondence corr(f_w, f_r) exists for any two $(f_w, f_r) \in S_w \times S_r$ if an object o exists such that

$$o \in O(f_w) \cap O(f_r) \text{ and} \tag{6}$$

$$t^s(f_r) \in [t^s(f_w), t^e(f_w)]. \tag{7}$$

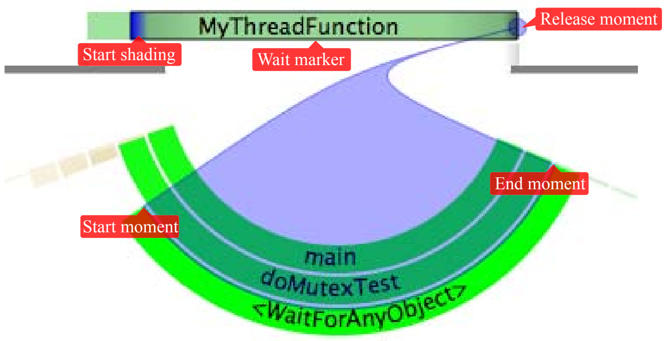


Fig. 3. Detailed view of a single correspondence curve.

In general, a function call may wait on or release more than one object. In those cases, a correspondence curve is rendered if wait and release call share at least one object. Wait calls on multiple objects may be classified into those that wait for all objects and those that wait for any. The first kind returns if all objects are released, and the latter kind returns as soon as a single object is released. To differentiate them, we render a distinct function name for each. Note that in the following text we often refer only to a single object instead of a set of objects to simplify explanations.

The correspondence curve is rendered as a shape with three anchor points (Fig. 3). Two of them mark the start and end waiting moments of f_w, and the third marks the release moment in f_r. In addition, a circle highlights the release moment, and the correspondence – on its way from one thread trace to the other – is drawn towards the circle center. Together, these result in an asymmetric curve shape that allows for identifying who waits on and who releases an object. The curve is painted in the color $\mathrm{col}(T_w)$, where T_w is the waiting thread trace ($f_w \in T_w$). This enables us to distinguish correspondence curves originating from different thread traces. Further, a curve is rendered semi-transparent to make it easier to follow them in case of crossings.

Additionally, a wait marker (see Fig. 3) is rendered on top of the releasing thread trace T_r, where $f_r \in T_r$. It shows the waiting time range $[t^s(f_w), t^e(f_w)]$ as a semi-transparent overlay bar on T_r. This enables us to examine what happens in T_r while f_w is waiting for an object owned by T_r.

Since wait markers of different thread traces may overlap, it is not clear anymore where a wait marking starts. To overcome this ambiguity, we render a *start shading* (see Fig. 3) in the color of the waiting thread ($\mathrm{col}(T_w)$). This aids in keeping track of the waiting trace while exploring the area surrounding the start of the wait marker. This is especially helpful when the release moment is outside of the viewport and, thus, many release markers overlap at the edge of the viewport.

2) Spatial Aggregation: We next describe two spatial aggregation strategies that we use to implement the multiscale design. We first discuss how to aggregate icicle plots and subsequently how we aggregate correspondence curves.

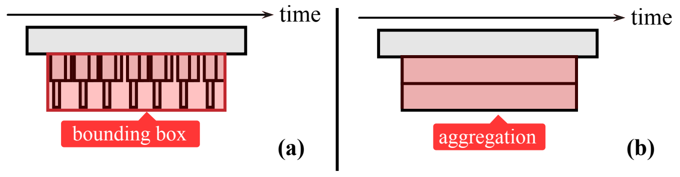


Fig. 4. Spatial aggregation strategy for icicle plots: (a) Bars which are too small to display, and their corresponding bounding box; (b) the replacement cluster indicating the aggregation.

Icicle-Plot Aggregation: For large traces, especially in zoomed-out mode, many short-duration function calls would be too small to be rendered accurately, and eventually cause moiré effects. Moreover, rendering many such (sub-)pixel sized bars has a significant performance impact without delivering added value. We therefore reduce the amount of bars to draw by aggregating smaller ones (less than a few pixels) by the following spatial aggregation strategy: We first compute clusters of very small and adjacent bars. Next, for each cluster a bounding box is computed, where the box encloses all bars in the cluster (Fig. 4a). Finally, a replacement cluster is rendered: For every level that the bounding box surrounds (Fig. 4b), we draw a bar with the horizontal extents of the bounding box. These bars are rendered in a lighter color to differentiate them from bars representing single function calls. In addition, with increasing depth their color becomes even lighter.

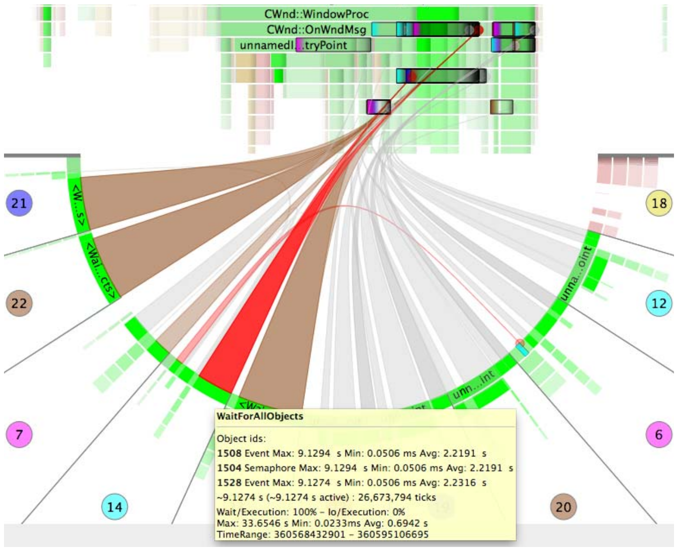
Correspondence-Curve Aggregation: To avoid overloading the visualization, it is further necessary to aggregate synchronization correspondences, too. Therefore, correspondence curves for aggregated synchronization calls f with $S(f) = S_s$ are rendered differently. We consider an aggregation cluster A as the set of all function calls in such a cluster. For every thread trace $T_m \in P$ we compute

$$\mathrm{rel}_A(T_m) = \{t^s(f_r) | f_r \in T_m \wedge f_w \in A \wedge \mathrm{corr}(f_w, f_r)\}. \quad (8)$$

Here, $\mathrm{rel}_A(T_m)$ can be interpreted as the set of all release moments that should be drawn to T_m from the function calls in A. Now, for each thread T_m a compound correspondence-curve is rendered. It consists of a filled path from the aggregation start moment to its end moment, the maximum release moment, and the minimum release moment back to the start.

The appearance of such path's ends, though, is asymmetric on purpose to keep such correspondences' wait side distinguishable from their release side: On the release side of the path, the curve is bended slightly to the circle center, thereby forming a concave shape. To prevent the aggregated paths' larger area (compared to their non-aggregated counterparts) to draw too much attention, we render the non-aggregated correspondences opaquely on the canvas' background, below their non-aggregated counterparts.

3) Attribute Mapping: All function calls in the icicle plots are rendered in one of three hues (green, red and brown). A function call is colored brown if neither it nor any of its descendants is doing I/O or waiting for an object. In contrast, a call f which performs I/O (synchronization) operations or



```
WaitForAllObjects
Object ids:
1508 Event Max: 9.1294  s Min: 0.0506 ms Avg: 2.2191  s
1504 Semaphore Max: 9.1294  s Min: 0.0506 ms Avg: 2.2191  s
1528 Event Max: 9.1274  s Min: 0.0506 ms Avg: 2.2316  s
~9.1274 s (~9.1274 s active) : 26,673,794 ticks
Wait/Execution: 100% – Io/Execution: 0%
Max: 33.6546 s Min: 0.0233ms Avg: 0.6942 s
TimeRange: 360568432901 – 360595106695
```

Fig. 5. In object color mode, hovering over a wait call shows correspondences that refer to the same objects.

has any descendants doing so, is colored depending on the ratio $w_r(f) = \frac{\text{dur}^o(f)}{\text{dur}^w(f)}$: For $w_r(f) < 0.5$, it is colored red, and green otherwise.

The intensity of green/red is proportional to $\frac{\text{dur}^w(f)}{\text{dur}(f)}$, $\frac{\text{dur}^o(f)}{\text{dur}(f)}$ respectively. That is, calls that exclusively wait or perform I/O are rendered with the highest intensity.

For correspondence curves, we provide two mapping modes to emphasize specifics of multithreaded behavior. The first mode, thread color mode, colors correspondence curves according to the color of the waiting thread T (see, e.g., Fig. 1). This helps finding who waits on whom and who releases locks for whom. The second mode, object color mode, highlights shared synchronization objects which are waited on in *multiple* threads. These emphasize and point to relationships between such two waiting threads (Fig. 5).

In this mode, an object's correspondences are grayed out if only a single thread waited on that object, ever. In contrast to aggregations, however, they still have an outline and are semi-transparent. Upon hovering over a wait call for a shared object, the following color mapping is applied to the respective correspondences:

A correspondence is colored

1) red, if it belongs to a wait call that operates on the same set of objects as the hovered one,
2) brown, if its wait call shares at least one object with the hovered call,
3) and gray otherwise

C. Interaction

After loading a program trace, thread traces can be selected from the selection window (see Fig. 1). Every newly selected thread trace is then displayed in the main view and any previously displayed thread trace is moved to the context view.

Both, the main and the context view are equipped with pan and zoom, so the viewport for every thread trace can be set individually. Furthermore, the main thread trace is reflected in the overview bar in a focus+context fashion.

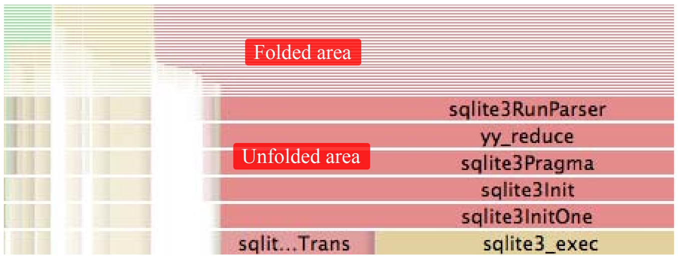
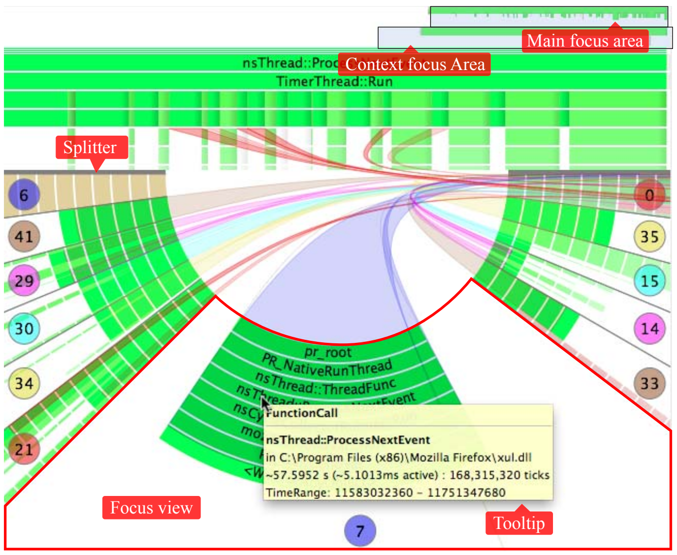


Fig. 6. Stack *folding*: Panning upwards successively shrinks upper stack levels one-by-one (folding); panning downwards successively resets the size of upper stack levels to normal one-by-one (unfolding)



```
FunctionCall
nsThread::ProcessNextEvent
in C:\Program Files (x86)\Mozilla Firefox\xul.dll
~57.5952 s (~5.1013ms active) : 168,315,320 ticks
TimeRange: 11583032360 – 11751347680
```

Fig. 7. 'Focusing' a thread-trace in the context view: The section of one selected thread trace on the semicircle is temporarily increased for more detailed analysis.

Instead of standard vertical scrolling, we *fold* the levels of a thread trace by panning towards the top (Fig. 6) and *unfold* them by panning towards the bottom. We basically apply a graphical fisheye view [31] with a single-step magnification function where the step is moved by panning. That is, by folding, stack levels of the icicle plot successively shrink in height, beginning from the top. So, panning upwards successively shrinks upper stack levels one-by-one to a minimum size; panning downwards successively resets the size of upper stack levels to normal one-by-one. This enables us to see the bottom of very deep thread traces without scrolling while maintaining the high-level context.

To allow for an on-demand partitioning of the screen space, main view and context view are separated by a vertically movable splitter. Also the zero-level radius is adjustable to either increase the space for showing correspondences or to show more stack levels in the context view. Further, a thread trace in the context view may be focused to give it more space than the others (Fig. 7). For that thread trace an overview

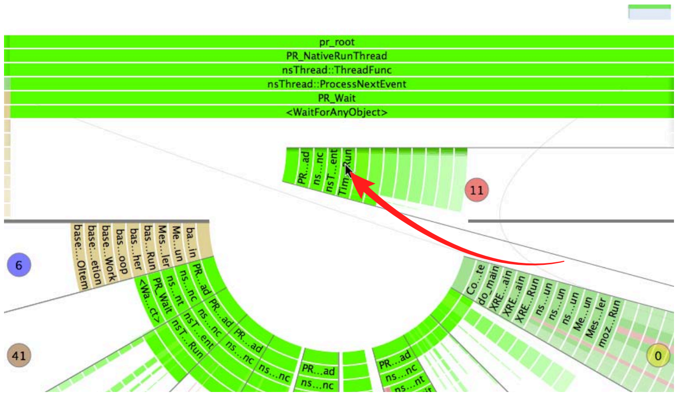
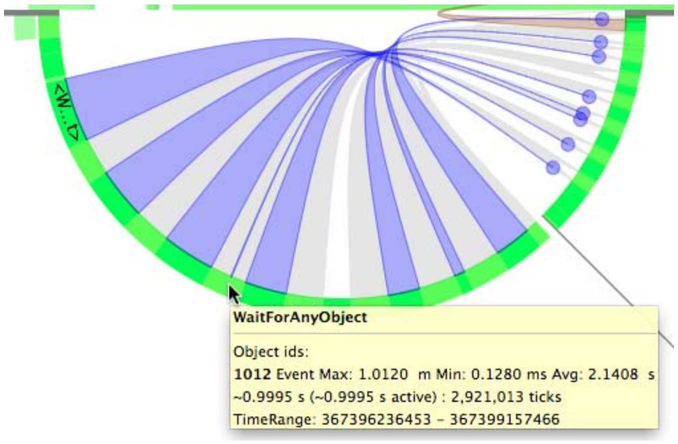


Fig. 8. Thread traces can be re-ordered according to the users' needs.



WaitForAnyObject
Object ids:
1012 Event Max: 1.0120 m Min: 0.1280 ms Avg: 2.1408 s
~0.9995 s (~0.9995 s active) : 2,921,013 ticks
TimeRange: 367396236453 – 367399157466

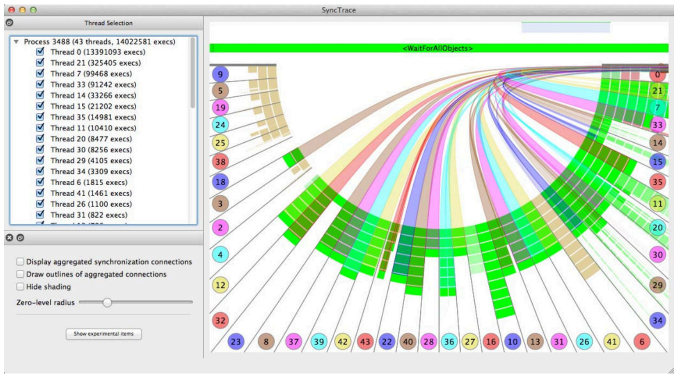
Fig. 9. The tooltip reveals which function calls are contained in the aggregation. In addition, the associated correspondence of the hovered call is displayed.

version is rendered in the overview bar and the focus area is indicated as for the main view.

To reorder thread traces we can drag&drop a trace on top of another. Those two traces are then swapped (Fig. 8). Thread traces can be reordered from context view to focus view or within the focus view to different sections on the semicircle. In case we want to quickly filter out several thread traces, we double click on the thread-id label to hide the corresponding thread trace.

When we hover over a function call a tooltip with detailed information is shown. In case we hover over an aggregation, the function call under the cursor is estimated and detailed information is presented, too. In addition, whenever the call is a wait call, the associated correspondence is rendered (Fig. 9).

In general, correspondences are only rendered if at least a part of the corresponding wait call is in the viewport. This includes correspondences whose release moment is not in the viewport. Those correspondences are rendered more transparent and the circle around the release moment is omitted. However, it is possible that a wait marker is still visible while the release marker is outside of the visible time range. To still see which correspondence belongs to which wait marker, the



Process 3488 (43 threads, 14022581 execs)
Thread 0 (13391093 execs)
Thread 21 (325405 execs)
Thread 7 (99468 execs)
Thread 33 (91242 execs)
Thread 14 (33266 execs)
Thread 15 (21202 execs)
Thread 35 (14981 execs)
Thread 11 (10410 execs)
Thread 20 (8477 execs)
Thread 30 (8256 execs)
Thread 29 (4105 execs)
Thread 34 (3309 execs)
Thread 6 (1815 execs)
Thread 41 (1461 execs)
Thread 26 (1100 execs)
Thread 31 (822 execs)

Display aggregated synchronization connections
Draw outlines of aggregated connections
Hide shading
Zero-level radius

Fig. 10. Initial program trace overview after loading all thread traces.

end of the correspondence is kept in the same stack level as the wait marker. Here, the missing circle indicates, that the end of the correspondence doesn't mark the release time. While the wait marker can leave the viewport on the other side as well, users can notice that by the missing start-shading (see Fig. 3).

If *all* aggregated correspondences would be rendered (opaque and without borders) for many selected thread traces, the resulting image would be unreadable due to visual clutter. Therefore, aggregated correspondences are rendered only for the thread trace which the mouse is currently hovering (detail on demand). In fact, only the correspondences on which function calls in the thread trace had to wait are visible.

V. Applications

We explain the usage of SyncTrace with the help of a program trace captured from the web-browser Firefox[1] (version 12.0.1) while it loads and renders a page of search results from Google. After removing the most often called 5190 functions, the remaining trace consists of approximately 14 million function calls in 43 threads. For all threads about 2700 synchronization and I/O calls were recorded. If all 149,869 functions were instrumented, the time required for loading the results page varied in between 15 to 33 seconds (8 seconds without I/O); without instrumentation, loading the page took 1-2 seconds. Each setting was executed 5 times. Instrumentation (with I/O) slowed the execution by a factor of 7 to 15. We therefore conclude that I/O is largely responsible for the massive 15 times slowdown, since execution slowdown without I/O was almost constant throughout our measurements. For a discussion of these overheads and possible solutions, see Sec. VI.

After loading the trace and displaying all thread traces at once, we see an overview representation of the synchronization dependencies (Fig. 10). We see that some threads don't wait for synchronization primitives at all (threads 9, 5, 19, 24, etc.), whereas many of the other threads are waiting for a long and continuous period (threads 3, 12, 32, ..., 35, 7). We can deduce that, because only one wide correspondence curve is rendered for the latter threads. Further, the visualization shows that all

[1]https://www.mozilla.org/firefox/, last accessed 07/11/2013

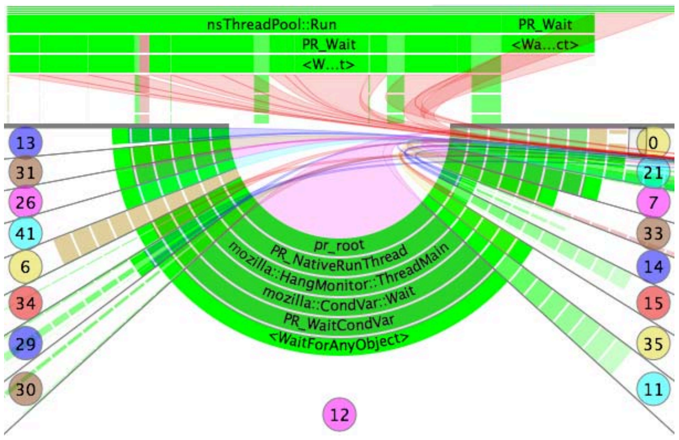


Fig. 11. Defunct hang monitor in Firefox: The hang monitor waits during its entire lifetime for an object to be released.

major wait calls are released by thread 0. This is an indication for a farmer/worker pattern, with thread 0 being the farmer. If so, we can tell by the long continuous wait calls, that many workers are mostly idle, and, hence, that too many workers were spawned initially.

Since some of the threads expose interrupted wait periods, they actually did some work in between. If there is in fact a farmer/worker pattern present and all waiting threads are workers, this indicates a workload imbalance among them.

To verify the farmer/worker hypothesis we examine the stack trace of those threads. For this, we drag one thread after another into the focus view to examine its activity and function names in more detail. Some of the threads execute *nsThread::ThreadFunc*, which in turn calls *nsThread::processNextEvent*. So, there are in fact threads that just do work item processing, and which have no other dependencies.

However, we found other kinds of threads. One of them drew our attention, since it executed *mozilla::HangMonitor::ThreadMain*, but was waiting for thread 0 to release it during its entire lifetime (Fig. 11). The function's name indicated that it is responsible for observing the rest of the application and to come into action if the software was hung. However, if the application would actually be hung, nobody could notify (i.e., wake up) the thread, so it is crucial that it wakes up periodically by itself to check whether the application is still responsive. By hovering over the waiting function call, the tooltip revealed that this thread is not waiting for a timer, but a semaphore. This rules out the possibility that we were simply not patient enough to see the thread wake up, i.e., the thread could not wake up by itself. Finally, by examining source code of *HangMonitor* we found that the wake up timeout was set to infinity and, thus, the hang monitor was effectively disabled.

Another thread (15) always called *js::SourceCompressorThread::internalCompress* after waiting for thread 0. Obviously, this is a dedicated Javascript compression thread. Since it always waited for thread 0, it

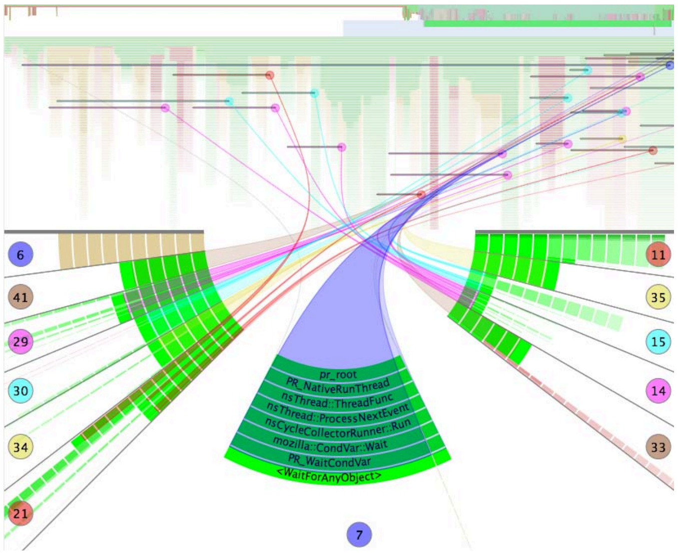


Fig. 13. Call stack of thread 0 folder for an overview. Release calls are made on varying stack levels.

seems to process compression tasks generated by thread 0. We expected a compression task not to do any synchronization or I/O. This hypothesis was supported by the brown shading of the compression functions, which would have been colored green or red otherwise.

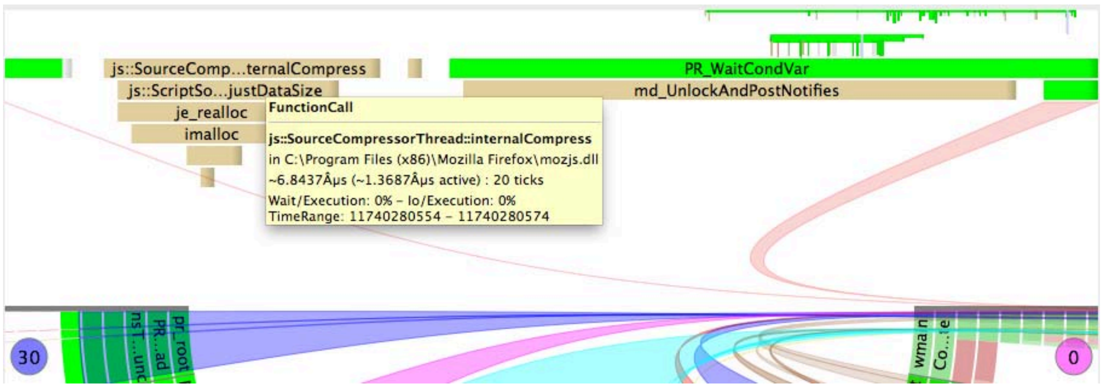
Since we did not load a Javascript-rich website, the compressor thread was mostly unoccupied and in many cases the compression tasks took less time than notifying thread 0 of the result (Fig. 12). This mostly idle thread suggests a waste of operating system resources. In addition, its very short task processing times may be too short to actually speed up the application or might even slow it down compared to a single threaded version.

Next, we focused on thread 0 since all of the other threads' long wait calls are released by it. To get an overview, we dragged thread 0 to the main view and folded all stack levels. If there were a loop, for example, we expected to see that releases would be made in the same stack levels, resulting in visible patterns. However, the resulting image did not show such patterns (Fig. 13).

This, in contrast to our hypothesis, suggests that thread 0 processes many separate (i.e., independent) tasks throughout its lifetime. So, the distribution of release moments, together with the variable stack depth of releases may be indicators for the behavioral complexity of the thread.

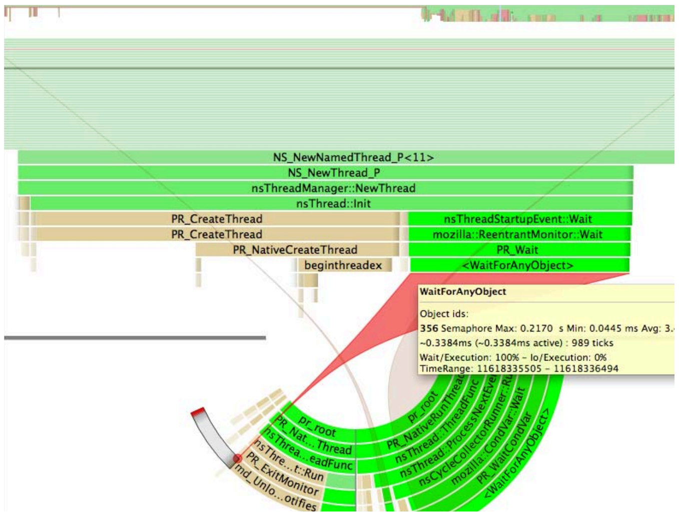
Since there are no visible wait correspondences going from thread 0 to another thread, thread 0 spent the majority of its lifetime computing instead of waiting. This is supported by the fact that only a few sections of its activity are colored green and if they are, they are only slightly saturated. Hovering over the thread's start function reveals that only 0.5% of its lifetime is spent waiting.

We recall that function calls which are influenced by blocking calls are highlighted in red or green, and the intensity



FunctionCall
js::SourceCompressorThread::internalCompress
in C:\Program Files (x86)\Mozilla Firefox\mozjs.dll
~6.8437Âµs (~1.3687Âµs active) : 20 ticks
Wait/Execution: 0% – Io/Execution: 0%
TimeRange: 11740280554 – 11740280574

Fig. 12. A (mostly idle) worker thread responsible for compressing Javascript.



WaitForAnyObject
Object ids:
356 Semaphore Max: 0.2170 s Min: 0.0445 ms Avg: 3.
~0.3384ms (~0.3384ms active) : 989 ticks
Wait/Execution: 100% – Io/Execution: 0%
TimeRange: 11618335505 – 11618336494

Fig. 14. Thread creation: The creator has to wait for the synchronous call to return.

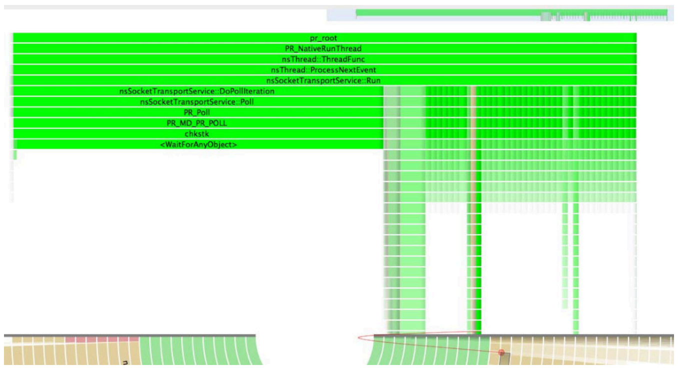

pr_root
PR_NativeRunThread
nsThread::ThreadFunc
nsThread::ProcessNextEvent
nsSocketTransportService::Run
nsSocketTransportService::DoPollIteration
nsSocketTransportService::Poll
PR_Poll
PR_MD_PR_POLL
chkstk
<WaitForAnyObject>

Fig. 15. Repetitive pattern of polling calls, strongly variable call duration.

indicates the amount of blocking. The green color of *do_main* signals that its descendants had to wait somewhere for a synchronization object.

To find out which parts of thread 0 are influenced by blocking calls (calls that could lead the scheduler to suspend the thread for a while) we follow the green function bars (wait calls) downwards (increasing stack level). By panning and zooming to the region of interest, aggregations reveal their content and previously aggregated correspondences are rendered individually. This enables us to follow single correspondences. For example, we can see that creating a new thread via *nsThreadManager::NewThread* implied waiting for the new thread to leave a monitor (synchronization) Fig. 14.

In addition to synchronization calls, I/O operations may block a thread, too. To find out more about the software portions that perform I/O, we follow the red colored functions bars as we did for the green ones before. We find that much of the I/O operations are performed by the SQLite database layer, where, in the most cases, I/O accounts only for a small fraction of the time spent for such database operation. However, we found an intensely red-colored aggregated activity that took approx 500ms to finish. The call stack reveals that it belongs to *nsDocShell::AddURIVisit*. Here, the tooltip further revealed that *AddURIVisit* takes around 0.1ms on average, only a tiny fraction of this 500ms outlier. Investigating other occurrences of *AddURIVisit* reveals that it normally performs only short-lived I/O operations or none at all. Consequently, calling *AddURIVisit* is sometimes a costly operation in terms of runtime, although, in the most cases, it is not. Since this call happened in the user interface thread, we suspect it was as cause for rendering Firefox temporarily unresponsive.

By dragging thread 10 to the main view we see a repetitive call pattern (Fig. 15). Zooming in a bit to read the function names, shows that this thread is polling every second. The stack pattern resembles that on the left of the visible part of the trace. However, while a single poll iteration took a minute, the frequency has now increased to a poll every second. This indicates that the poll interval was reduced in between.

To reveal object sharing among threads, we switch into the object color mode (see Sec. IV-B3). Fig. 16 shows that all connections are greyed out, which indicates that Firefox uses separate synchronization objects for every thread.

This is in contrast to, for example WordPad, that makes excessive use of shared objects (see Fig. 5).

The visual analysis outlined above took about 15 minutes. We note the added value of the stack folding technique, which

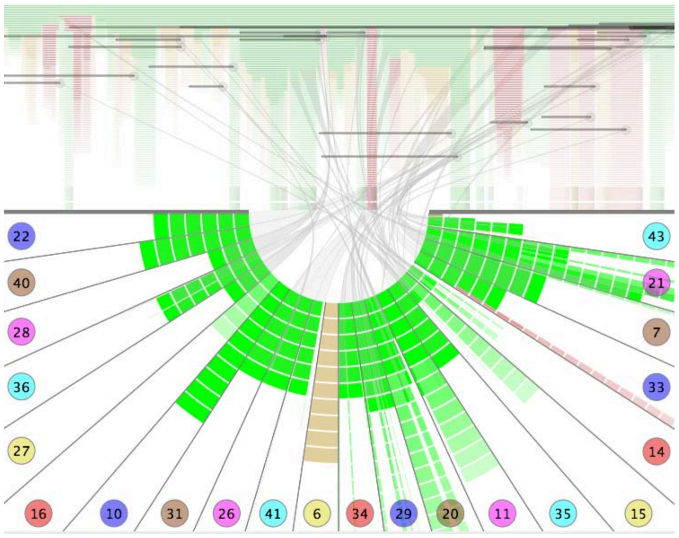


Fig. 16. Object color mode in the Firefox program trace: Since all connections are grayed out every thread has its own synchronization objects to wait on.

enabled us to quickly gain an overview in which levels in the call stack synchronization objects are released. The start shading for function bars enabled to find small-scale repetitive patterns which we would have missed otherwise. Despite some crossings in the correspondence curves, we were able to discern the threads' individual correspondences with the help of our interaction techniques and the provided attribute mappings.

VI. Discussion

Generality: While we demonstrate our approach on large program traces, it can be used to visualize any given set of hierarchical sequences with correlations. The correlations are not restricted to leaf nodes, i.e., they can be one-to-one matches on any hierarchy level.

Visual Scalability: Our enhanced spatial layout enables the parallel depiction of many threads' activities and inter-correspondences between those threads. We reduce visual clutter by specific interaction and multiscale aggregation techniques. This lets us visually analyze hundreds of thousands of calls and correspondences in multiple thread traces in an interactive manner.

Ease of Use: The zoom&pan interaction techniques enable users to freely adjust the shown subsequences and level of detail. Drag&drop can be used to quickly re-define the focused thread as well as to reposition threads in the context view. Moreover, removing uninteresting threads from the current analysis set is as easy as double clicking their visual representation.

Limitations: With the visual design being more flexible than CodeFlows (we can show correspondences between any two trees) and more scalable than TraceDiff (we can show more than two trees), it as well has its limits in terms of very large traces or massively multithreaded traces.

Due to the way we extract wait correspondences, we are limited to analyzing blocking system calls. In contrast, library calls on already-released objects, which are non-blocking, can thus not be analyzed.

Practicality: In general, the execution overhead imposed by tracing may influence the thread-interplay and program behavior and can also vary per thread depending on the number of instrumented functions and function calls. Also, as seen in Sec. V, slow (trace) disk I/O can interfere with the execution. To reduce this influence, the pintool uses only non-blocking calls. More importantly, while dynamic instrumentation, which we use, is more versatile than static instrumentation, it is also known to be significantly slower at runtime. Hence, tracing overhead can be much lower in scenarios where a less versatile tracing solution is acceptable. Nevertheless, and in particular for very short execution durations where the tracing overhead is relatively high compared to the original execution duration, the measured durations may be misleading. For instance, highly active threads, that execute many calls, might execute slower than less active threads doing less calls. If such less active thread waits on a highly active thread, this could lead to more or longer waiting calls in the less active thread. This can be partially alleviated by excluding such low-level utility calls from tracing [5], [40].

VII. Conclusions

We presented SyncTrace, a visualization technique for the analysis of dependencies between threads of execution in concurrent programs. We combine common straight and bended icicle plots in a hybrid juxtaposed view to implement a focus+context approach. By this, we are able to depict more threads of execution in the same screen space than existing techniques. Moreover, by the juxtaposition, we can use the center of the viewport to depict relationships between any two threads of execution as multiscale edge bundles. We use attribute mapping to colors and shape of the edge bundles to encode important runtime meta-data.

As future work, we will examine different designs of the edge bundles to depict additional attributes. Furthermore, we intend to evaluate the benefit of additional interaction techniques such as gestures.

References

[1] C. Artho, K. Havelund, and S. Honiden. Visualization of concurrent program executions. In *Proceedings of the Annual International Computer Software and Applications Conference*, pages 541–546, Washington, DC, USA, 2007. IEEE Computer Society.

[2] F. Beck, R. Petkov, and S. Diehl. Visually exploring multi-dimensional code couplings. In *Proceedings of the International Workshop on Visualizing Software for Understanding and Analysis*, pages 1–8. IEEE, Sept. 2011.

[3] M. Bedy, S. Carr, X. Huang, and C.-K. Shene. A visualization system for multithreaded programming. *SIGCSE Bulletin*, 32(1):1–5, 2000.

[4] J. Berthold and R. Loogen. Visualizing parallel functional program runs: Case studies with the eden trace viewer. In *Proceedings of the International Conference Parallel Computing*, pages 121–128, 2007.

[5] J. Bohnet. *Visualization of Execution Traces and its Application to Software Maintenance*. PhD thesis, Hasso-Plattner-Institute at the University of Potsdam, Germany, 2010.

[6] T. A. Corbi. Program understanding: challenge for the 1990's. *IBM Systems Journal*, 28(2):294–306, 1989.

[7] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[8] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang. Visualizing the execution of java programs. In *Revised Lectures on Software Visualization, International Seminar*, pages 151–162, 2002.

[9] S. Diehl. *Software Visualization. Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, Berlin, 2007.

[10] S. Eichelbaum, M. Hlawitschka, and G. Scheuermann. LineAO — improved three-dimensional line rendering. *IEEE Transactions on Visualization and Computer Graphics*, 19(3):433–445, March 2013.

[11] J. Fiedor and T. Vojnar. Noise-based testing and analysis of multi-threaded C/C++ programs on the binary level. In *Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, pages 36–46, New York, NY, USA, 2012. ACM.

[12] S. Fleming, E. Kraemer, R. Stirewalt, and L. Dillon. Debugging concurrent software: A study using multithreaded sequence diagrams. In *Proceeding of the Symposium on Visual Languages and Human-Centric Computing*, pages 33–40. IEEE, 2010.

[13] M. Fowler. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley Professional, 2000.

[14] M. Graham and J. Kennedy. A Survey of Multiple Tree Visualisation. *Information Visualization*, 9:235–252, 2009.

[15] A. Hamou-Lhadj. *Techniques to Simplify the Analysis of Execution Traces for Program Comprehension*. PhD thesis, University of Ottawa, 2005.

[16] M. T. Heath and J. A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8:29–39, 1991.

[17] D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 12:741–748, 2006.

[18] D. Holten and J. J. Van Wijk. Visual comparison of hierarchically organized data. *Computer Graphics Forum*, 27(3):759–766, 2008.

[19] J. C. D. Kergommeaux, B. D. O. Stein, and M. S. Martin. Pajé: An extensible environment for visualizing multi-threaded program executions. In *European Conference on Parallel Computing*, pages 133–144, 2000.

[20] B.-C. Kim, S.-W. Jun, D. J. Hwang, and Y.-K. Jun. Visualizing potential deadlocks in multithreaded programs. In *Proceedings of the International Conference on Parallel Computing Technologies*, pages 321–330. Springer-Verlag, 2009.

[21] J. B. Kruskal and J. M. Landwehr. Icicle plots: Better displays for hierarchical clustering. *The American Statistician*, 37(2):162–168, 1983.

[22] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the Conference on Programming language design and implementation*, pages 190–200. ACM SIGPLAN, 2005.

[23] K. Mehner. *Trace-based Debugging and Visualisation of Concurrent Java Programs with UML*. PhD thesis, Universität Paderborn, 2005.

[24] S. Moreta and A. Telea. Multiscale visualization of dynamic software logs. In *Proceedings of the Eurographics Conference on Visualization*, pages 11–18, 2007.

[25] T. Munzner, F. Guimbretière, S. Tasiran, L. Zhang, and Y. Zhou. Treejuxtaposer: scalable tree comparison using focus+context with guaranteed visibility. *ACM Transactions on Graphics*, 22(3):453–462, July 2003.

[26] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. Vampir: Visualization and analysis of mpi resources. *Supercomputer*, 12:69–80, 1996.

[27] G. Nutt, A. Griff, J. Mankovich, and J. McWhirter. Extensible parallel program performance visualization. *International Symposium on Modeling, Analysis, and Simulation of Computer Systems*, 0:205–211, 1995.

[28] D. L. Parnas. Software aging. In *Proceedings of the International Conference on Software Engineering*, pages 279–287, 1994.

[29] S. P. Reiss. Visualizing the java heap. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*, pages 251–254, New York, NY, USA, 2010. ACM.

[30] J. Roberts. TraceVis: An Execution Trace Visualization Tool. In *Proceedings MoDS*, pages 123–130, 2005.

[31] M. Sarkar and M. H. Brown. Graphical fisheye views. *Communications of the ACM*, 37(12):73–83, Dec. 1994.

[32] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2000.

[33] J. Stasko and E. Zhang. Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations. In *Proceedings of the Symposium on Information Visualization*, pages 57–65. IEEE, 2000.

[34] J. T. Stasko. The parade environment for visualizing parallel program executions: A progress report. Technical report, Georgia Institute of Technology, 1995.

[35] M.-A. D. Storey, F. D. Fracchia, and H. A. Mueller. Cognitive design elements to support the construction of a mental model during software visualization. In *Proceedings of the 5th International Workshop on Program Comprehension*, pages 17–28, Washington, DC, USA, 1997. IEEE Computer Society.

[36] A. Telea and D. Auber. Code Flows: Visualizing Structural Evolution of Source Code. *Computer Graphics Forum*, 27(3):831–838, 2008.

[37] A. Telea and O. Ersoy. Image-Based Edge Bundles: Simplified Visualization of Large Graphs. *Computer Graphics Forum*, 29(3):843–852, 2010.

[38] J. Trümper, J. Bohnet, and J. Döllner. Understanding Complex Multithreaded Software Systems by Using Trace Visualization. In *Proceedings of the International Symposium on Software Visualization*, pages 133–142. ACM, 2010.

[39] J. Trümper, J. Döllner, and A. Telea. Multiscale Visual Comparison of Execution Traces. In *Proceedings of the International Conference on Program Comprehension*, 2013.

[40] J. Trümper, S. Voigt, and J. Döllner. Maintenance of Embedded Systems: Supporting Program Comprehension Using Dynamic Analysis. In *Proceedings of the International Workshop on Software Engineering for Embedded Systems*, pages 4396–4402, 2012.

[41] L. Voinea, A. Telea, and J. J. van Wijk. EZEL: a Visual Tool for Performance Assessment of Peer-to-Peer File-Sharing Networks. In *Proceedings of the Symposium on Information Visualization*, pages 41–48. IEEE, 2004.

[42] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44 –55, Aug. 1995.

[43] K. Wheeler and D. Thain. Visualizing massively multithreaded applications with threadscope. *Concurrency and Computation: Practice and Experience*, 22:45–67, 2009.

[44] S. Xie, E. Kraemer, and R. E. K. Stirewalt. Design and evaluation of a diagrammatic notation to aid in the understanding of concurrency concepts. In *Proceedings of the International Conference on Software Engineering*, pages 727–731, Washington, DC, USA, 2007. IEEE.

[45] A. Zaidman. *Scalability Solutions for Program Comprehension Through Dynamic Analysis*. PhD thesis, Unviversiteit Antwerpen, 2006.

[46] O. Zaki, E. Lusk, and D. Swider. Toward scalable performance visualization with jumpshot. *High Performance Computing Applications*, 13:277–288, 1999.

[47] Q. A. Zhao and J. T. Stasko. Visualizing the execution of threads-based parallel programs. Technical report, Georgia Institute of Technology, 1995.

[48] M. Zinsmaier, U. Brandes, O. Deussen, and H. Strobelt. Interactive level-of-detail rendering of large graphs. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2486–2495, 2012.